"PRETTY GOOD SOFTWARE PROTECTION"

fluid studios

IMPORTANT

If you are reading this document, you are expected to be a shareware author that I personally trust. This trust means that you are free to use this scheme as you see fit, provided you keep this information secret.

If you have received this information from an individual other than me, please email me at `midnight@FluidStudios.com`. Thanks.

Finally, this document uses C/C++ for example purposes, though the technique is viable to any language.

TABLE OF CONTENTS

INTRODUCTION

This key-based protection scheme differs from any other protection scheme in a very fundamental way. Rather than a simple *validation* scheme, the key itself becomes part of the application; without a valid key, the application is incapable of running.

This does not mean that the key contains code. On the contrary, the key contains data; data required to allow the application to run. It can be implemented by any language, requires no self-modifying code, or any other non-standard uses of a language.

This document is intended to explain how the protection scheme works in general. Each implementation by various shareware programmers using this scheme will differ, but they will all be based on the same underlying methodology.

Before we get into the details, I'd like to make an important point. The goal of this protection scheme is not to be completely, 100% "crack-proof" (personally, I don't believe this is possible.) Rather, the goal is to create a scheme that requires more time than any would-be cracker is willing to spend on the problem. Trust me on this: if the cracker enjoys the challenge, there is virtually no time limit.

We can further increase our protection by producing a scheme that can be changed over time, while still using the same underlying methodology for generating keys. This means that we can completely change how we use the registration key, but not how we generate them. To be clear: our key generation won't change; our first key is generated in exactly the same way as our most recent key, yet by changing the way we access the key, it becomes a moving target for any would-be cracker.

If we wish, we can do this each time we release a new version of our software. It's not necessary, but it only takes just a few minutes and is a good idea.

Therefore, we have a more clearly defined goal: To produce a protection scheme that will take the would-be cracker longer to crack the scheme than it takes us (the software developers) to update our software.

THEORY BEHIND THE KEY

Before we get into the technical details, let's first cover what the key is and how it works. This section will cover the basic idea in its simplest form. Make no mistake; a key generated for practical use is not this simple…

In essence, the key contains a series of simple math formulae. For example, one formula might be `a + b = 12`.

Consider the following 16 bits:

```
0000100100000011
```

If we extract the first 8 bits `[00001001 = 9]` into the variable `a` and extract the second 8 bits `[00000011 = 3]` into `b`, then our formula becomes `[9 + 3 = 12]`. In this case, our formula is correct, because `a + b` does, indeed, equal `12`.

How do we apply this to our code? Consider the following macro (again, simplified for explanation purposes):

```
#define strlen(x) (strlen(x) + a + b - 12)
```

Note that because of the ANSI C/C++ preprocessor rules, this will actually work. After all, these rules were setup so you could do things just like this. If you're uncomfortable with this, this code could just as well be placed in an `inline` function. The advantage to using `inlines` and/or macros is to have a single place where they key is used within the source code, which propagates throughout the entire program's object code. Therefore, anybody examining a disassembly of the executable will see hundreds (or thousands) of places where the key is accessed.

So let's look at this macro. We place a call to `strlen`, which calculates the length of the string `x`, we then add `a` and `b`, which results in `12`, and then we subtract `12`. If the key is a correct key, the result of `[a + b - 12]` will equal zero, returning the proper result from the `strlen`. However, if the key is invalid, our application may very well crash or result in other undesired behavior.

In fact, the only way this application will run, is if the portion of the key (containing the bits for `a` and `b`) is valid.

You might think that this method of populating the key would slow an application down. In reality, I've found that this is not true, if you are careful. Using `strlen` is a great example, as it is not a function you're likely to call in time-critical code. Also, the extra code required to check the key is much faster than the `strlen` function itself. Therefore, you're guaranteed to only suffer a negligible performance hit.

By now, you should have a good idea of the fundamental concept behind how the key works.

If we stop here, we've done a very poor job at protecting our application. Why? Because anybody with the smallest amount of experience can disassemble the application, figure out how each part of the key works, reverse engineer it and create a program to generate keys (i.e., a *keygen*.)

So we're going to extend our key in a number of ways which will be discussed in the next few sections.

OVERVIEW OF THE REGISTRATION KEY

As you'll see, you'll have a lot of freedom on exactly how you generate your registration keys. The format of the key is up to you. I personally chose a 128-bit key with the following format:

80-bits for the calculated key data
32-bits for the unique id
16-bits for the key checksum

For the sake of explanation, we'll simplify our key to be only 32-bits (exactly 1/4th as many bits as I've chosen for practical use.) You can, of course, extend this to be a key of any size you wish. Thus, our example key format becomes:

```
xxxxxxxxxxxxxxxxxxxx yyyyyyyy zzzz
```

...where x represents one bit of calculated key data (total of 20 bits), y represents one bit of the unique ID (total of 8 bits) and z represents a bit of checksum data (total of 4 bits.)

It's important to note that with an 8-bit value for the unique ID field, we'll only allow for 256 unique keys. In practical use, a 32-bit unique ID would be preferred.

The next few sections will explain how these different fields are calculated. I'll warn you now, that the following explanation might be a bit long winded. This is because the key itself is obfuscated to help hide its true nature from any would-be crackers and/or keygen makers.

A MORE PRACTICAL KEY

In a previous section, we discussed the use of simple mathematical formulae performed on bits extracted from the key. Now we're going to extend this concept by placing a number of formulae in the key. In order to have a number of formulae, we'll need a number of variables.

Shared-bit and Unique-bit Variables

In order to make a key that is difficult to reverse-engineer, we're going to use two kinds of variables: *shared-bit variables* and *unique-bit variables*.

To help understand what a *shared-bit variable* is, let's look at the following example:

```
        10011101
mask a: a-a-a-a-
mask b: --bb-b-b
```

Here, we have an 8-bit value (`10011101`) and the bit masks for two variables (a & b). These are not your normal bit masks; rather, they depict which four bits of an 8-bit pattern each variable occupies.

Each variable contains only four bits. So, looking at the four bits for a, we find it uses every other bit (starting from the left) which results in the bit pattern: `1010`. Similarly, the bit pattern for b is: `0111`.

Consider the following table of *shared-bit variables*:

```
Key bits   : xxxxxxxxxxxxxxxxxxxx yyyyyyyy zzzz

mask for a:                       a-a-a-a-
mask for b:                       --bb-b-b
mask for c:                       cc--c--c
mask for d:                       dd--d-d-
mask for e:                       --ee-e-e
mask for f:                       -f-f-ff-
```

The top line (`Key bits`) simply shows our example key's bit format (as explained in the previous section.) Remember that x denotes the calculated bits, y denotes the unique ID and z denotes the checksum. In this table, we're only interested in those 8 bits that are used for the unique ID.

Below our key bits, are the masks for the variables a - `f`. These masks are arbitrarily chosen, I've just typed them in by hand (just like you will.) There are a few constraints, though, that you'll need to adhere to.

Each variable requires four bits (as per our example key) and these bits are localized to the portion of the key containing the unique ID. If you look closely at the bit masks for each of the variables a - `f`, you'll notice that they each occupy four bits and that many of these bits are shared amongst other variables. Thus, variables a – `f` are *shared-bit variables*.

It is important that the bits of the shared-bit variables be fairly evenly distributed throughout the unique ID portion of the key. To further demonstrate this in our example, consider the same table with a single column highlighted:

```
mask for a:                    a-a-a-a-
mask for b:                    --bb-b-b
mask for c:                    cc--c--c
mask for d:                    dd--d-d-
mask for e:                    --ee-e-e
mask for f:                    -f-f-ff-
```

The highlighted column shows that the third bit (from the left) is shared by exactly three variables. If you look closely, you can see that every bit (column) is shared by exactly three variables. This isn't absolutely necessary; just know that the more evenly distributed your bits are, the more difficult it will be to reverse-engineer.

Let's move on to the *unique-bit variables*. Consider the following table:

```
Key bits   : xxxxxxxxxxxxxxxxxxxx yyyyyyyy zzzz

mask for g: gggg----------------
mask for h: ----hhhh------------
mask for i: --------iiii--------
mask for j: ------------jjjj----
mask for k: ----------------kkkk
```

It should be clear that the variables g – `k` are also 4-bit variables, occupy a different area of the key (the *calculated key data* area) and that they share no bits. These *unique-bit variables* are important, because they make it possible to generate the key.

Much of the key generation process is comprised of arbitrarily chosen specifications (i.e. the bit masks for the shared bit variables, the math formulae, etc.) This is important for a key that is very difficult to reverse engineer, because there are few patterns. Later, you'll see how we can obfuscate those patterns even more.

So let's talk about generating that key.

PREPARING TO GENERATE A KEY

The first thing we'll need is a set of arbitrary bit masks for our variables. As stated in the previous section, these masks are generated by hand. So let's go back to our example masks and make one large table (note that this table is now complete – each mask representing the bit positions for a variable within a full 32-bit key value, including placeholders for the checksum portion of the key):

```
a: --------------------a-a-a-a-----
b: ----------------------bb-b-b----
c: --------------------cc--c--c----
d: --------------------dd--d-d-----
e: ----------------------ee-e-e----
f: ---------------------f-f-ff-----
g: gggg----------------------------
h: ----hhhh------------------------
i: --------iiii--------------------
j: ------------jjjj----------------
k: ----------------kkkk------------
```

When writing the keygen, we'll use these exact tables to help create our keys. Here's how:

```
static const char *mask_a = "--------------------a-a-a-a-----";
static const char *mask_b = "----------------------bb-b-b----";
static const char *mask_c = "--------------------cc--c--c----";
static const char *mask_d = "--------------------dd--d-d-----";
static const char *mask_e = "----------------------ee-e-e----";
static const char *mask_f = "---------------------f-f-ff-----";
static const char *mask_g = "gggg----------------------------";
static const char *mask_h = "----hhhh------------------------";
static const char *mask_i = "--------iiii--------------------";
static const char *mask_j = "------------jjjj----------------";
static const char *mask_k = "----------------kkkk------------";
```

The C strings shown above are our bit masks. More specifically, each string represents which bits (from a 32-bit value) each of our 4-bit variables will occupy. In order to extract those bits from a 32-bit word, we'll need a helper function that looks like this:

```
// Extract the bits from a 32-bit ‘key’ using the string ‘mask’

unsigned char getKeyBits(const char *mask, const unsigned int key)
{
        unsigned char result = 0;
        unsigned int  bitOffset = 0;

        // Visit each bit in the mask (we go backwards so we can
        // pack the bits in the proper order)

        for (int i = 31; i >= 0; --i)
        {
                // Skip unoccupied bits in the mask

                if (mask[i] == '-') continue;

                result += ((key >> i) & 1) << bitOffset;
                ++bitOffset;
        }

        return bits;
}
```

We’ll also need a routine that does the exact opposite; rather than extract bits from a key, it will need to place bits into the key. This is left as an exercise to the reader.

Important note:

Until now, we’ve been dealing with 4-bit variables and 32-bit keys. As stated in an earlier section, a 32-bit key with an 8-bit unique ID simply won’t do; and that I’ve chosen a 128-bit key for practical use. It is also important that you are aware that using 4-bit variables is also not ideal. This is because the math formulae we use will use mod-256 math (i.e. `255 + 1 = 0`), which we’ll accomplish by placing our variables into `unsigned chars`. As you will find in later sections, this will be important when it comes time to use the key within the application. This also drastically changes the routine shown above (`getKeyBits`) as well as its counterpart.

We will, however, continue using our simplified 32-bit key with 4-bit variables for the sake of a simplistic explanation.

GENERATING THE KEY

We start by calculating a unique ID for the key. In the case of our example, this is an 8-bit value. In my software, I do this by calculating a random value. As the name for this field ("Unique ID") suggests, the value must be unique. In order to validate this, you'll need to check it against a database of other unique IDs you've generated for other customers.

This ID is the only thing that makes the key unique. If two keys were to be calculated with the same ID, then the two keys generated would be identical. Therefore, the unique ID can serve as a customer identifier.

When the 32-bit key is populated with the unique ID in the proper bits, you'll find that our shared-bit variables are given arbitrary values. We have no control over what's in these values, yet we will still use them in our math formulae.

Before we go further, let's recap

We have our 32-bit key which is comprised of 20-bits of "calculated data", 8-bits for a unique ID and 4 bits for a key checksum. We also have a set of bit masks that denote how the bits within the key are distributed amongst a series of 4-bit variables. These masks will be the same for every key generated. With these bit masks, we have a total of 11 variables (some of which share bits with other variables, and some of which share no bits.) The unique ID is calculated and placed in the 32-bit key, which assigns arbitrary values to our shared-bit variables. These variables (both shared-bit and unique-bit) are to be used in a set of simple math formulae.

Let's talk about those math formulae

Here are some arbitrarily chosen math formulae we'll use for an example:

```
a + g     = 12
b – h     = 5
c + i     = 15
d - j     = 13
e + f – k = 9
```

Much like the masks, these formulae are arbitrarily chosen. Once chosen, they are hard-coded into the key generation routine, so that every valid key will adhere to the same set of formulae.

Note that since our example deals with 4-bit variables, our math must be performed mod-16 (i.e. `a + b & 0x0F`).

Remember that some of our variables are shared-bit variables, and some are unique-bit variables. In our specific example, we have 6 shared-bit variables (`a - f`) and 5 unique-bit variables (`g - k`). Also, it is safe to assume that we have virtually no control of the contents of the shared-bit variables, as they share bits with the unique ID (chosen at random) and with one another. So, how can we be expected to create a series of formulae that will always result in the same constant value across every key? We use the unique-bit variables, of course!

Let's take another look at those arbitrary formulae once again:

```
a + g     = 12
b – h     = 5
c + i     = 15
d - j     = 13
e + f – k = 9
```

These formulae may be arbitrary, but they also conform to a constraint: each formula contains at least one unique-bit variable. We can then use simple algebra to calculate these values. For example, the first formula listed is [`a + g = 12`]. Because `a` comes from a few bits of our unique ID, we don't really have control over what it is. However, once we've calculated our unique ID, we can extract `a` from the key to find out what value it does contain. If `a` were to contain the value `6`, then we simply solve the equation:

```
 6 + g = 12
12 - 6 = g
     g = 6
```

Because we must use at least one unique-bit variable with each formula, the maximum number of formulae chosen may not be more than the number of unique-bit variables you have within your key. In our example, we have five unique-bit variables, which allow us a maximum of five formulae.

A quick note about mod-256 math

We need the use of mod-256 math for our formulae because this ensures that we can properly solve each formula. For example, given the formula [`8 - b = 9`] requires mod-256 math to solve for `b`.

We can now finish calculating our key. All we need to do is solve each equation to calculate the unique-bit variables. Given our formulae listed above, here is the solution to each of the unique-bit variables:

```
// extract the shared-bit variables

     a = getKeyBits(mask_a, key);
     b = getKeyBits(mask_b, key);
     c = getKeyBits(mask_c, key);
     d = getKeyBits(mask_d, key);
     e = getKeyBits(mask_e, key);
     f = getKeyBits(mask_f, key);

     // Solve each equation for the unique-bit
     // variables

     g = 12 - a;
     h =  5 + b;
     i = 15 - c;
     j = 13 + d;
     k =  9 - (e+f);
```

With our unique-bit variables properly calculated, we can populate our key with their bits.

At this point, we're nearly done. We've calculated our unique ID (which occupies 8 bits of the key), and we've solved for our unique-bit variables, which occupies 20 bits of our key. We're left with only four bits: the checksum.

You can calculate the checksum in any way you wish. The goal of the checksum is to allow you to determine, at a glance, if a key is *potentially valid*. This simply allows you to recognize when a user has entered an invalid registration key. Four bits (as in our example) won't make for a very good checksum – be sure to use at least 8 or 16 bits in practice.

DO NOT validate your key by checking each of the formulae! This gives a would-be cracker an immediately clear look at how your key is built. Instead, we're going to require that the would-be cracker go through a lot more effort to crack your software.

OBFUSCATION, ANYBODY?

We've already performed our first task of obscuring the nature of the key. And that is by using the shared-bit variables. The beauty of these shared bits is that if any single bit is changed, then at least 3 shared-bit variables are damaged. With larger keys come more variables and more bits to be shared among a number of these shared-bit variables.

To further complicate matters, the same applies for the unique-bit variables. There is an important point that deserves some attention here. Because the formulae must always result in exactly the same value, if any single bit is changed in either of the two groups of variables (shared-bit or unique-bit) then the **key is damaged beyond repair**. I'd like to further point out just how badly damaged the key is…

Just how hard is this key to reverse engineer?

First consider what happens when we change a bit from the shared bit group. It should be clear that toggling any bit in this group results in a minimum of three variables being damaged. If you think carefully about the ramifications of changing one bit in the shared-bit group, you'll find that this destroys three variables at a minimum. In order to correct this problem, the non-shared bits must be adjusted to accommodate. However, modifying those bits would further complicate matters by forcing the would-be cracker to have to adjust even more bits within the shared-bit grouping. This cycle repeats and never ends. This is because there is only **ONE** way to build a key. That is to know exactly how the key is built (including all formulae), start with the shared-bit variables, and then use algebra to calculate the unique-bit variables. Trying to build the key in any other order would prove futile.

Second, consider what happens if we change a single bit from the unique bit group. This may only affect a single formula but correction is virtually impossible, because that would mean adjusting at least one shared-bit variable, which, in turn would cause other shared-bit variables to change, and the cycle repeats itself yet again.

Therefore, it is safe to say that if any bit in any part of the working key (shared-bit or unique-bit groups) is modified, the key is very badly damaged.

At this point, it should be clear that creating a valid key requires 100% knowledge (including the bit format, the formulae, etc.) of how the key is built via reverse engineering of the code that uses the key. This would prove to be a very difficult task, but not impossible. So let's now consider how we can further obfuscate the key. Can we make it impossible? Almost!

We'll start by obfuscating the masks. Consider the following set of masks:

```
static const char *mask_a = "---a------------a---a------a----";
static const char *mask_b = "-b------b--------------b---b----";
static const char *mask_c = "---c----c--c--------c-----------";
static const char *mask_d = "---d-------d----d---d-----------";
static const char *mask_e = "-e------e--------------e---e----";
static const char *mask_f = "-f---------f----f------f--------";
static const char *mask_g = "g---g-------g-------------g-----";
static const char *mask_h = "------h---h-----------h-h-------";
static const char *mask_i = "-------------iii---------i------";
static const char *mask_j = "-------j---------jj--j----------";
static const char *mask_k = "--k--k---k---------k------------";
static const char *maskID = "-x-x----x--x----x---x--x---x----";
```

These are the same masks as those we started with. The only difference is that I manually shuffled the columns around. If you look at the first column, you'll see only one variable occupies it, (g, a unique-bit variable.) Looking at the second column, you'll find that three variables occupy it (our shared-bit variables.) As you can see, all the data is present, it's just been reordered.

I was careful to leave the last four columns in place, because we'll want our checksum to always occupy the last bits of our key. You may have noticed that I've also added an extra mask: `maskID`. This is necessary, because after we've shuffled our key around, we don't know which bits represent our unique ID. We just need to use this mask when populating our key with the unique ID (so we know which bits to set) and our obfuscation is complete.

A more realistic, obfuscated key

In a more practical example, the keys used in my software were 128 bits wide and contained 26 variables (a real mess!) Here's an example of a valid set of masks using a 112-bit key similar to the key used in my software:

```
----------------a---------a--a-------------a----------------------a---------a-------------a-------a-------------
---------bb--------------------bb--------bb-------------------------------b---------b---------------------------
-----c-----------------------c-------------cc----c-------c--c--------------c------------------------------------
---------------------d-dd-----------d--------------d-----------------dd-----------------------------------d-----
---ee------------e---------ee-e----------------------------------------------------e-------------e--------------
-----------f--------------------------------------f-------ff-----f-----------ff--------------------------f------
-------------------------------------------------------g------------g-----------g-g----------ggg-------g--------
---hh---------------------hhh-h----h-h--------------------------------------------------------------------------
-------ii-------------i-----------i----------iii---------------------------------i------------------------------
----------------------------jj-------j--------------------j------j-----------j------------j------j--------------
------------k--k----------------------kk--------k---k----------------------------------kk-----------------------
-----l---------------------l--l------l----------------------------l----------l------------l--------------l------
---m-------------m--------m-------------------------------mm---------------m-m-------------------m--------------
--------------n----nn------------------------------------------nn--------n----------------------n----------n----
-----o-----------o---------o--------------------------------o----oo------------------------------o-------o------
-----------p-----------------p-------p------p----p--------p-------p-----------p---------------------------------
-----------------------------------------------------qq------------q-----------q------------q-----------q---qq--
----r------r--------------r----------------------r---------r----------------r------r--------------r-------------
------------------s---------------------s--------------------ss--------ss------------------s------------------s-
-----t------------------------t-------------t-----t------t-------t------------t-------------------t-------------
----------------uu-----------------u--------u----u---------u------------------------------u--------------u------
------v-------------------------------------------------v----------------------------vv--v---------vvv----------
----w------------------------------w-------w------w---------w--------------ww------w----------------------------
---x------------x------------------x-------x------x------x--x---------------x-----------------------------------
-----------y----y-----------y----------------------------y-----------------y--y----y--------------y-------------
zzz----------z-----------z-------z--------------------------------------------------------------------z--------z
```

We can obfuscate the key further by not using all of the math formulae in the code. This offers two advantages: First, by not placing all of the formulae in the code, the would-be cracker doesn't have all the information necessary to create a whole key. Second, we can switch which formulae are used (still holding back a few) in each release of the software, so that our key becomes a moving target for any would-be cracker, without ever giving them all the information required to create a complete key.

Also, since our method of populating the code with key-validation is through commonly-used `inline` functions and macros, the would-be cracker would need to find every instance where we use the key (potentially thousands of individual instances) in order to be absolutely certain that he has all the information available, and if we never use every formula, the would-be cracker still wouldn't have enough information to produce a fully valid key.

If we find that a keygen was actually produced, we could safely assume that the keygen was created without all the information (because we never let them see all the formulae.) We can combat this keygen by switching the variables & formulae used in the next release. This would break the keys produced by a keygen, yet still allow customer keys to work properly. Because of this, be sure to design your keys with some breathing room of extra variables and formulae.

Finally, with the mass population of key-accesses throughout the code, it might be wise to hide one or two accesses in inconspicuous places. Set aside one or two formulae for use in a single, critical place in the code. Those individual key formulae will be surrounded by hundreds or thousands of key accesses, so finding them will be like finding a needle in a very large haystack.

You might think that the would-be cracker could simply place a breakpoint on a memory-read for a portion of the key to locate those hard-to-find formulae, but if you've properly randomized your key, those bits will be spread out across multiple bytes, shared with the bytes of other formulae.

USING THE KEY IN YOUR CODE

Done properly, this should be very straight forward. In my case, my entire protection scheme resides in header files. The magic is done entirely with macros. I'll walk you through each group of macros, and show you some practical code.

This first macro is used as the basis for picking out 8-bit variables from my 128-bit key. The macro receives 8 parameters, each of which represents the bit index into the key (0-127), and packs the resulting bits into a single 8-bit value:

```
#define      key_var(v0,v1,v2,v3,v4,v5,v6,v7) \
             ((((key[(v0>>3)]<<(v0&7))&0x80)>>0) |   \
              (((key[(v1>>3)]<<(v1&7))&0x80)>>1) |   \
              (((key[(v2>>3)]<<(v2&7))&0x80)>>2) |   \
              (((key[(v3>>3)]<<(v3&7))&0x80)>>3) |   \
              (((key[(v4>>3)]<<(v4&7))&0x80)>>4) |   \
              (((key[(v5>>3)]<<(v5&7))&0x80)>>5) |   \
              (((key[(v6>>3)]<<(v6&7))&0x80)>>6) |   \
              (((key[(v7>>3)]<<(v7&7))&0x80)>>7))
```

These next two macros use the previous to reference a single variable from within the key. They simply contain the hard-coded indices. These indices are calculated from the mask strings, so that we don't need to actually include the mask strings with our application. Otherwise, the would-be cracker might find it a little too easy to figure things out. Note that these are only the first two key variables (of my 26.)

```
#define      key_a        key_var(1, 53, 64, 65, 77, 83, 99, 102)
#define      key_b        key_var(5,  6, 32, 39, 40, 47, 61,  78)
```

Next, we have a set of macros that represent the key functions. Here are two of mine:

```
#define      keyfunc_0    (unsigned char)(key_a - key_b)
#define      keyfunc_1    (unsigned char)(key_c + key_d)
```

Following that, I have a set of values that define what each key formula should result in. Here are the first two:

```
#define      keyres__0    0x39
#define      keyres__1    0x2A
```

Next, I've chosen to obfuscate the process even further. I use these macros when calculating key results. Note that sometimes, when I use a formula, it actually combines two formulae in a single block, making the key that much harder to decipher. Here's a sampling of only two “extended” functions (with this methodology, you really can create a nice plethora of possible key uses):

```
#define fubarfunc_0 ((keyfunc_0+keyfunc_2)-(keyres__0+keyres__2))
#define fubarfunc_1 ((keyfunc_7          )-(keyres__7          ))
```

And finally, the last step is to propagate the use of the key throughout the entire code base. Here’s a sampling of how mine are propagated:

```
#define      memset(a,b,c)       memset((a),(b) | fubarfunc_1,(c))
#define      toupper(a)          toupper((a) + fubarfunc_7)
#define      tolower(a)          tolower((a) + fubarfunc_8)
```

You can create as many of these as you wish. Just be careful to use them with functions that will take considerably longer to calculate than the overhead introduced by the key calculations. Also, you want to use functions that are used frequently in your program.

After propagating the keys throughout my code, I found that my executable considerably. After analysis, it turns out that my key calculations were 33% of my executable size, with no noticeable performance degradation!

However, every application has different needs, so in the next section, I’ll discuss how to make this protection scheme manageable within a project, given real-world concerns (like maintenance.)

SIMPLIFYING MAINTENANCE OF PROTECTED CODE

You certainly don't want to have to try to debug code that has been protected in this way! Also, if you're like me, you want as little modification to your existing code base as possible. I've solved these problems by using three header files.

The first header file `Fubar.h` contains all the macros shown above with the exception of the final set of macros (those macros that actually cause the compiler to propagate the key accesses throughout your code.)

These propagation macros were placed in a file titled `Fubar_on.h`. Finally, I created a third header file called `Fubar_off.h`, which disables the propagation macros. They look like this:

```
#undef        memset
#undef        toupper
#undef        tolower
```

For the most part, you'll want the protection propagated throughout your entire code base. To do this, include both `Fubar.h` and `Fubar_on.h` in a common header file that is included in every source file (such as `stdafx.h`, if you are using MFC.)

For those files that contain time critical code or that you do not want protection to propagate through, simply include `Fubar_off.h` at the top.

In some cases, you may have a file that you want to propagate with key accesses, with the exception of a single routine. To do this, simply include `Fubar_off.h` before the routine, and then include `Fubar_on.h` after it.

Finally, you may also wish to conditionally discard the protection. My particular project contains many configurations, only one of which is the final released product (a release mode build of the registered version of the software.) In this case, I've added these lines to `Fubar_on.h` so that the key is only propagated through that specific build:

```
#if    defined(_REGISTERED_)

... contents of file ...

#endif // _REGISTERED_
```

With these three headers, you can safely and effectively control exactly where and how the key is propagated through your code base.

FINAL WORDS

In order to properly protect your software, you should produce two separate versions: a trial version and a registered version. Not doing so would result in a program that is well protected, but still very easily cracked, because the would-be cracker would only need to crack the trial software to re-enable the disabled code.

Your keys, as presented to the user, may take any form. I chose a format similar to:

897D-45A0-338B-BDDB-8D3A-4557-312A-B4DE

Others have used ASCII formats similar to the following, which can be used to represent the same number of bits:

A4FJE8G05H28T4YK

I prefer the longer hex format, simply because it may appear to be more intimidating. <g>

There are many other topics to discuss, such as server-side checks, etc. I have decided to stop here, having explained the technique to the best of my ability. I hope it's clear enough. If not, I will assume that you have received this document directly from me, in which case, you are free to email me to ask questions.

Thanks for reading!

- Paul Nettle
- MidNight@FluidStudios.com